Exam 1 Review

Topics

Reasoning about code
- Forward and backward reasoning, logical conditions, Hoare triples, weakest precondition, rules for assignment, sequence, if-then-else, loops, loop invariants, decrementing functions

Forward Reasoning
- Forward reasoning simulates the execution of code. Introduces facts as it goes along
  
  E.g., \( \{ x = 1 \} \)
  
  \[ \begin{array}{l}
  y = 2 \times x \\
  \{ x = 1 \land y = 2 \} \\
  z = x + y \\
  \{ x = 1 \land y = 2 \land z = 3 \}
  \end{array} \]
  
  Collects all facts, often those facts are irrelevant to the goal

Backward Reasoning
- Backward reasoning “goes backwards”. Starting from a given postcondition, finds the weakest precondition that ensures the postcondition

  E.g., \( \{ y < 1 \} \) // Simplify \( 2y < y+1 \) into \( y < 1 \)
  
  \[ \begin{array}{l}
  z = y + 1 \\
  \{ 2^2y < z \} \\
  x = 2^2y \\
  \{ x < z \}
  \end{array} \]
  
  More focused and more useful

Condition Strength
- “P is stronger than Q” means “P implies Q”
- “P is stronger than Q” means “P guarantees more than Q”
  
  E.g., \( x > 0 \) is stronger than \( x > 1 \)

  Fewer values satisfy P than Q
  
  E.g., fewer values satisfy \( x > 0 \) than \( x > 1 \)

  Stronger means more specific
  
  Weaker means more general
Condition Strength

- Which one is stronger?
- \( x > -10 \) or \( x > 0 \)
- \( x > 0 && y = 0 \) or \( x > 0 || y = 0 \)
- \( 0 \leq x \leq 10 \) or \( 5 \leq x \leq 11 \) (Neither!)
- \( y \equiv 2 \text{ (mod 4)} \) or \( y \) is even
- \( x = 10 \) or \( x \) is even

Hoare Triples

- A Hoare Triple: \( \{ P \} \text{ code } \{ Q \} \)
- \( P \) and \( Q \) are logical statements about program values, and \( \text{code} \) is program code (in our case, Java code)
- \( \{ P \} \text{ code } \{ Q \} \) means “if \( P \) is true and we execute \( \text{code} \), then \( Q \) is true afterwards”
- \( \{ P \} \text{ code } \{ Q \} \) is a logical formula, just like “\( \text{0sindex} \)”

Examples of Hoare Triples

- \( \{ x > 0 \} \ x++ \{ x > 1 \} \) is true
- \( \{ x > 0 \} \ x++ \{ x > -1 \} \) is true
- \( \{ x = 0 \} \ x++ \{ x > 1 \} \) is false. Why?

- \( \{ x > 0 \} \ x++ \{ x > 0 \} \) is ??
- \( \{ x < 0 \} \ x++ \{ x < 0 \} \) is ??
- \( \{ x = a \} \text{ if } (x < 0) \ x = -x \{ x = |a| \} \) is ??
- \( \{ x = y \} \ x = x + 3 \{ x = y \} \) is ??

Rules for Backward Reasoning: Assignment

- // precondition: ??
- \( x = \text{ expression} \)
- // postcondition: Q
- **Rule:** the weakest precondition = Q, with all occurrences of \( x \) in Q replaced by \( \text{expression} \)
- More formally:
  \( \text{wp}("x=\text{expression};",Q) = Q \) with all occurrences of \( x \) replaced by \( \text{expression} \)

Rules for Backward Reasoning: Sequence

- // precondition: ??
- \( S1; \) // statement
- \( S2; \) // another statement
- // postcondition: Q
- Work backwards:
  - **precondition** is \( \text{wp}("S1;S2;",Q) = \text{wp}("S1;","\text{wp}("S2;","Q)) \)
- Example:
  - // precondition: ??
  - \( x = 0; \)
  - \( y = x+1; \)
  - // postcondition: \( y > 0 \)
  - // precondition: ??
  - \( x = 0; \)
  - \( y = x+1; \)
  - // postcondition: \( y > 0 \)

Rules for If-then-else

- **Forward reasoning**
  \( \{ P \} \)
  \( \{ (b \wedge \text{wp}("S1","Q)) \lor (\neg b \wedge \text{wp}("S2","Q)) \} \)
- if \( b \)
  \( \{ \text{wp}("S1","Q)) \}
- \( S1 \)
- \( \{ Q \} \)
- else
  \( \{ \text{wp}("S2","Q)) \}
- \( S2 \)
- \( \{ Q \} \)
- \( Q \)
Reasoning About Loops by Induction

1. Partial correctness
   - Guess loop invariant
   - Prove loop invariant using computation induction
   - Loop exit condition (i.e., loop condition at false) and loop invariant imply the desired postcondition

2. Termination
   - (Intuitively) Guess “decrementing function” D.
   - Each iteration decrements D, D = 0 AND loop invariant, imply loop exit condition (i.e., D = 0 AND loop invariant imply loop condition evaluates to false)

Example: Reasoning About Loops

Precondition: x >= 0; i = x; z = 0;
while (i != 0) {
    z = z + 1;
    i = i - 1;
}
Postcondition: x = z;
Need to prove:
1. x = z holds after the loop (partial correctness)
2. Loop terminates (termination)

1) i=x and z=0 give us that i+z=x holds at 0th iteration of loop // Base case
2) Assuming that i+z=x holds after kth iteration, we show it holds after (k+1)th iteration // Induction
   z_{new} = z + 1 and i_{new} = i - 1 thus
   z_{new} + i_{new} = z + 1 + i - 1 = z + i = x
3) If loop terminated, we know i = 0.
   Since z+i = x holds, we have x = z
4) Loop terminates. D is i. D before > D after. D = 0 implies i = 0 (loop exit condition).

Topics

Specifications
- A specification is a contract between a method and its caller
  - Obligations of the method (implementation of specification): agrees to provide postcondition if precondition held!
  - Obligations of the caller (user of specification): agrees to meet the precondition and not expect more than postcondition promised
Example Specification

Precondition: \( len \geq 0 \) && \( arr.length = len \)
Postcondition: returns \( \text{arr}\[0\]+…+\text{arr}\[\text{arr}.length-1\]}

\[
\text{double sum(int\[\] arr, int len) \{}
\quad \text{double sum = 0.0;}
\quad \text{int i = 0;}
\quad \text{while (i < len) \{}
\quad \quad \text{sum = sum + arr\[i\];}
\quad \quad \text{i = i+1;}
\quad \}}
\text{return sum;}
\]

For our purposes, we will be writing specifications that are a bit less formal than this example. Mathematical rigor is welcome, but not always necessary.

Benefits of Specifications

- Precisely documents method behavior
- Imagine if you had to read the code of the Java libraries to figure what they do!
- An abstraction --- abstracts away unnecessary detail
- Promotes modularity
- Enables reasoning about correctness
- Through testing and/or verification

PoS Specifications

- Specification convention due to Michael Ernst
- The precondition
  - requires: clause spells out constraints on client
- The postcondition
  - modifies: lists objects (typically parameters) that may be modified by the method. Any object not listed under this clause is guaranteed untouched
  - throws: lists possible exceptions
  - effects: describes final state of modified objects
  - returns: describes return value

Example

\[
\text{static List<Integer> listAdd(List<Integer> lst1, List<Integer> lst2) \{}
\quad \text{List<Integer> res = new ArrayList<Integer>();}
\quad \text{for (int i = 0; i < lst1.size(); i++)}
\quad \quad \text{res.add(lst1.get(i) + lst2.get(i));}
\quad \text{return res;}
\}
\]

Another Example

\[
\text{static void listAdd2(List<Integer> lst1, List<Integer> lst2) \{}
\quad \text{for (int i = 0; i < lst1.size(); i++)}
\quad \quad \text{lst1.set(i, lst1.get(i) + lst2.get(i));}
\quad \}
\]

Specification Style

- A method is called for its side effects (effects clause) or its return value (returns clause)
- It is bad style to have both effects and return
- There are exceptions
  - E.g., HashMap.put returns the previous value
- Main point of spec is to be helpful
  - Being overly formal does not help
  - Being too informal does not help either
- If spec turns too complex: redesign. Better to simplify code than document complexity!
What's Wrong?

```java
static void uniquefy(List<Integer> lst) {
    for (int i = 0; i < lst.size()-1; i++)
        if (lst.get(i) == lst.get(i+1))
            lst.remove(i);
}
```

Specification Strength

- "A is stronger than B" means
  - For every implementation \( I \)
    - \( I \) satisfies \( A \) implies \( I \) satisfies \( B \)
    - The opposite is not necessarily true
  - For every client \( C \)
    - \( C \) works with \( B \) implies \( C \) works with \( A \)
    - The opposite is not necessarily true
- Principle of substitutability:
  - A stronger spec can always be substituted for a weaker one

Why Care About Specification Strength?

- Because of substitutability!
- Principle of substitutability
  - A stronger specification can always be substituted for a weaker one
  - i.e., an implementation that conforms to a stronger specification can be used in a client that expects a weaker specification

Substitutability

- Substitutability guarantees correct software updates, correct class hierarchies
- Client code: \( x \) \( x; \ldots x.foo(index); \)
- Client is "polymorphic": written against \( x \), but it is expected to work with any subclass of \( x \)
- A subclass of \( x \), say \( y \), may have its own implementation of \( foo \), \( y.foo(int) \). Client must work correctly with \( y.foo(int) \)
- If spec of \( y.foo(int) \) is stronger than that of \( x.foo(int) \) then we can safely substitute \( y.foo(int) \) for \( x.foo(int) \)!

Strengthening and Weakening Specification

- Strengthen a specification
  - Require less of client: fewer conditions in \texttt{requires} clause \texttt{AND/OR}
  - Promise more to client: \texttt{effects}, \texttt{modifies}, \texttt{returns}
- Weaken a specification
  - Require more of client: add conditions to \texttt{requires} \texttt{AND/OR}
  - Promise less to client: \texttt{effects}, \texttt{modifies}, \texttt{returns} clauses are weaker, thus easier to satisfy in code

Comparing by Logical Formulas

- \( (I \text{ satisfies specification } A) \) is a logical formula: \( P_A \Rightarrow Q_A \)
  - \( P_A \) is the precondition of \( A \), \( Q_A \) is the postcondition of \( A \)
- Spec \( A \) is stronger than spec \( B \) if and only if for each implementation \( I \), \( (I \text{ satisfies } A) \Rightarrow (I \text{ satisfies } B) \) which is equivalent to \( A \Rightarrow B \)
- \( A \) is stronger than \( B \) iff \( (P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B) \)
- Recall from FoCS and/or Intro to Logic: \( p \Leftrightarrow q \equiv \exists ! p \lor q \)
Comparing by Logical Formulas

if and only if

\[ P_B \rightarrow (Q_B \| P_A) \&\& [(P_B \&\& Q_A) \rightarrow Q_B] \]

A is stronger than B if and only if

\[ P_B \rightarrow P_A \&\& Q_A \rightarrow Q_B \] implies A is stronger than B

Exercise: Order by Strength

Spec A: requires: a non-negative int argument
returns: an int in [1..10]

Spec B: requires: int argument
returns: an int in [2..5]

Spec C: requires: true
returns: an int in [2..5]

Spec D: requires: an int in [1..10]
returns: an int in [1..20]

Converting PoS Specs into Logical Formulas

PoS specification

- requires: R
- modifies: M
- effects: E // absorbs throws, returns and effects

Spec is equivalent to the following logical formula:

\[ R \rightarrow (E \&\& (\text{nothing but } M \text{ is modified})) \]

Step 1: absorb throws and returns into effects \( E \)
Step 2: write \( R \rightarrow (E \&\& (\text{nothing but } M \text{ is modified})) \)

Convert Spec to Formula, step 1:

Absorb throws and returns into effects

- \( \text{from java.util.ArrayList\langle T\rangle} \)
- \( T \text{ set(int index, T element)} \)

requires: true
modifies: this[index]
effects: \( \begin{cases} \text{this}_\text{post}[\text{index}] = \text{element} & \text{if index} < 0 \lor \text{index} \geq \text{size} \\ \text{throws IndexOutOfBoundsException} & \text{if index} < 0 \lor \text{index} \geq \text{size} \\ \text{this}_\text{pre}[\text{index}] \end{cases} \)

Absorb effects, returns and throws into new effects:

if index < 0 || index \geq size then

\( \text{throws IndexOutOfBoundsException} \)

else this\_post[\text{index}] = \text{element} and returns this\_pre[\text{index}]

Denote effects expression by \( E \).

Resulting formula is:

\[ \text{true} \Rightarrow (E \land (\text{foreach } i \neq \text{index}, \text{this}_\text{post}[i] = \text{this}_\text{pre}[i])) \]

Topics

- ADTs
  - Benefits of ADT methodology, Specifying ADTs, Rep invariants, Representation exposure, Checking rep invariants, Abstraction functions
ADTs

- Abstract Data Type (ADT): higher-level data abstraction
  - The ADT is operations + object
  - A specification mechanism
  - A way of thinking about programs and design

An ADT Is a Set of Operations

- Operations operate on data representation
- ADT abstracts from organization to meaning of data
- ADT abstracts from structure to use
- Data representation does not matter!

```java
class Point {
    float x, y;
}
class Roint {
    float x, y;
}
```

Instead, think of a type as a set of operations: create, x(), y(), r(), theta().
Force clients to call operations to access data

Specifying an ADT

<table>
<thead>
<tr>
<th>immutable</th>
<th>mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>class TypeName</td>
<td>class TypeName</td>
</tr>
<tr>
<td>1. overview</td>
<td>1. overview</td>
</tr>
<tr>
<td>2. abstract fields</td>
<td>2. abstract fields</td>
</tr>
<tr>
<td>3. creators</td>
<td>3. creators</td>
</tr>
<tr>
<td>4. observers</td>
<td>4. observers</td>
</tr>
<tr>
<td>5. producers</td>
<td>5. producers (rare!)</td>
</tr>
<tr>
<td>6. mutators</td>
<td>6. mutators</td>
</tr>
</tbody>
</table>

Connecting Implementation to Specification

- Representation invariant: Object \(\rightarrow\) boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  - Defines the set of valid values

- Abstraction function: Object \(\rightarrow\) abstract value
  - What the data structure really means
    - E.g., array \([2, 3, -1]\) represents \(-x^2 + 3x + 2\)
  - How the data structure is to be interpreted

Representation Exposure

- Client can get control over rep and break the rep invariant! Consider
  ```java
  IntSet s = new IntSet();
  s.add(1);
  List<Integer> li = s.getElements();
  li.add(1); // Breaks IntSet's rep invariant!
  ```
- Representation exposure is external access to the rep. **AVOID!!!**
- If you allow representation exposure, document why and how and feel bad about it.

Make a copy on the way out:
```java
public List<Integer> getElements() {
    return new ArrayList<>(data);
}
```

- Mutating a copy does not affect IntSet's rep
  ```java
  IntSet s = new IntSet();
  s.add(1);
  List<Integer> li = s.getElements();
  li.add(1); // mutates new copy, not IntSet's rep
  ```
Representation Exposure

- Make a copy on the way in too:
  ```java
  public IntSet(ArrayList<Integer> elts) {
    data = new ArrayList<Integer>(elts);
    ...
  }
  ```
- Why?

Abstraction Function

- Abstraction function allows us to reason about correctness of the implementation

IntSet Example

- Creating concrete object:
  - Establish rep invariant
  - Establish abstraction function
- After every operations:
  - Maintains rep invariant
  - Maintains abstraction function